Designing a Reorder Buffer in Bluespec

Nirav Dave
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: ndave@mit.edu

Abstract

Production capabilities for complex VLSI chips have outpaced the ability of current generation CAD tools to design and verify such chips effectively. Bluespec is designed to synthesize high-level descriptions in the form of guarded atomic actions into high quality structural RTL. While much work has been done on verifying both the correctness and synthesizability of Bluespec descriptions, the work on realistic large scale designs is in early stages. This paper explores the design of the reorder buffer for an out-of-order superscalar processor with a MIPS I ISA. We discuss the design methodologies which are suited for large scale Bluespec design and discuss some of the difficulties we encountered. Even though the work is still in progress, we show what level of performance is achievable under the current Bluespec compiler and what problems need to be solved to make the tool viable for commercial production environments.

1. Introduction

Bluespec has been used at Sandburst, MIT and CMU to describe complex hardware. Previous work has also shown that small but complex designs described using TRS, the formalism underlying Bluespec, are amenable to formal verification [1]. It has also been shown that a simple 5-stage MIPS pipeline can be synthesized from TRS's quite efficiently [7, 8] . What remains to be seen is if the correctness-centric Bluespec design approach is able to generate RTL that is comparable to handwritten Verilog.

In this paper we present the Bluespec design process by designing a 2-way reorder buffer (ROB) to be used in a processor core for the MIPS I ISA. The focus of this paper is to determine whether this complex hardware design can be described in Bluespec such that its "cycle-level performance" is equivalent to what one would expect from handcrafted RTL written to support the concurrency of the tasks ROB has to perform. One needs to see if the high-level description captures the inherent concurrency of the design appropriately before delving into low-level timing and area optimizations.

1.1 Related Work

There is a lot of interest in high-level hardware description languages which make use of behavioral modeling, while still allowing for efficient hardware synthesis. Most commercial work in this field is focused on two approaches. The first is to increase the complexity level of RTL languages to be more suitable for modeling, such as Behavioral Verilog. The second is to modify a standard language (e.g. C or Java) to be more appropriate for describing hardware. In the latter case, Control Data Flow Graphs are extracted from the source description and techniques for compiling vector architectures are used to generate register transfer logic[4][6]. Neither of these techniques has yet to produce a widely accepted hardware description language for synthesis.

One type of research has a focus on specialized programmable processors[6][10]. This effort is only marginally associated with the problem of general purpose hardware description languages, as most of its emphasis is on processor specific issues, such as instruction encoding, and the automatically generated assemblers.

Two other types of languages have been explored by the research community to become an effective high-level HDL. The first of these types uses a synchronous specification language like Esterel, Signal, or Lustre. These languages deal with real-time issues[2]. Methods to compile Esterel into hardware have been written, but the results are not comparable to handwritten Verilog designs.

The second type uses an asynchronous language with atomic actions such as Dill's Murphi[3], Sere's Action Systems[9], Staunstrup's Synchronized Transitions[11],

0-7803-8509-8/04/$20.00 © 2004 IEEE

and Arvind and Shen's TRS[1]. The primary principle of these languages is that all hardware systems can be described in two parts: a physical state (e.g. registers and storage) and a set of guarded atomic actions which describe the state-change transitions. It has been shown that these atomic descriptions can be translated into efficient hardware if rules are assumed to take one cycle[7][8]. Bluespec is a member of this second group of languages.

1.2 Paper Organization

Section 2 of this paper gives a description of Bluespec's syntax and scheduling. Section 3 describes the rough design of the processor and the reorder buffer's function. Section 4 discusses the initial implementation of the reorder buffer in Bluespec. A discussion of the debugging process is detailed in Section 5. We write about optimizations done to improve cycle performance in Section 6, and other optimizations in Section 7. Finally, in Section 8 we discuss the findings of this work, general Bluespec design tips, and areas for future work for Bluespec.

2 Bluespec

Bluespec is an object oriented HDL which compiles into TRS. In Bluespec, a module is the actual unit which gets compiled into hardware. Each module roughly corresponds to a Verilog module. A module consists of three things: state, rules which modify that state, and interfaces which allow the outside world to interact with the module.

2.1 Bluespec Syntax

A module is the representation of a circuit in Bluespec. It can be a primitive module which is just a wrapper around an actual Verilog module, or a standard module with state elements including other modules, rules, and interface methods.

The state elements such as registers, flip-flops, memories are all specified explicitly in a module. The behavior of the module is represented by rules which each consist of a state change on the hardware state of the module (an *action*) and the conditions required for the rule to be valid (a *predicate*). It is valid to execute (*fire*) a rule whenever its predicate is true. The syntax for a rule is:

```
"RuleName":
 when predicate
  ==> action
```

The interface of a module is a set of methods through which the outside world interacts with the module. Each interface method has a predicate (a *guard*) which restricts when the method may be called. A method may either be a read method (i.e. a combinational lookup returning a value), or an action method, or a combination of the two, an actionValue method.

An actionValue is used when we do not want a value to be made available unless an appropriate action in the module also occurs. Consider a situation where we have a FIFO of values and a method that should get a new value on each call. From the outside of the module, we would want to be able to look at the value only when it is being dequeued from the FIFO. Thus we would write the following where do is used to signify an actionValue.

```
getVal = do
           fifoVal.deq
         return fifoVal.first
```

The abstract model of execution of a Bluespec circuit is as follows. For any initial hardware state, we have some set of executable rules. Each cycle, we randomly select one of these rules and execute it thereby changing the state. This is of course very inefficient, and so we allow multiple rules to fire at once, but require that any transition from one state to another must be obtainable by a valid sequence of single rule firings.

2.2 Scheduling

Due to the possible complexity of determining when a rule will use an interface of a module, Bluespec assumes conservatively that an action will use any method that it might ever use. That is to say that if the action accesses a method only when some condition is met, the scheduler will treat it as always using it. Using this simplification, the compiler scans the rules for two kinds of parallel operations on rule pairs. The first is conflict free, which means that each rule in the pair does not read what the other rule writes for either the predicate or the action, and the rules do not make the same method calls (e.g. writes). The second is sequential composition, which means that one rule does not read anything modified by the second and they do not both use the same methods.

These definitions miss some parallel operations. One rule may write to a state element in the others predicate, but not affect the predicate. In this case, the compiler incorrectly considers this a conflict. In general, there is no effective solution for this problem.

Describing the compiler's scheduling choices in detail is beyond the scope of this paper. For our purposes, it is enough to assume that we have some prioritizations on the sets of rules, where proper subsets of a set will have lower priority (i.e. the scheduling favors firing as many rules as possible). When a choice of which rule set must be made,

the rule set with the highest priority that can be fired will be chosen.

2.3 Verification

One of the key benefits of the Bluespec model is the ease of verification. The state change each cycle can be viewed as a sequential firing of rules. Thus, we can show a design is correct by verifying each rule is correct in isolation. The messy issue of concurrency is entirely handled for us by the Bluespec compiler.

RWires, an abstract wire module which we describe in more depth in Section 6 can cause the design to become sensitive to concurrency issues. However, we can easily handle this in our model as long as provide that actions which read from a RWire can always fire whenever an action which writes to that RWire can occur.

3 Design Considerations

In this section, we go over the high-level design of the processor and the roles of the subcomponents. We then discuss the performance requirements which our reorder buffer must meet.

3.1 Structure of the Processor

A reorder buffer contains decoded instructions in program order. It is responsible for determining when these instructions are executable, sending them to the appropriate functional unit, updating the state of the register file, and handling branch mispredictions.

We can view the processor abstractly as shown in Figure 1. Each unit must follow the following abstract requirements.

The Fetch/Decode Logic must send the ROB a string of decoded instructions in program order of a possible branch path. These instructions should be all tagged with an "epoch" value defined below. It also must contain an interface which the ROB can use to notify it of the new program counter (pc) and epoch whenever the ROB detects a branch misprediction.

The epoch is an integer value which is incremented on every branch miss. The ROB ignores all incoming instructions whose epoch values do not match the current value as they are part of the mispredicted path.

The ALU Unit must be able to take ready to execute tagged instructions from the ROB and execute those instructions. It must then eventually return each result with the associated tag of the instruction. No restrictions are placed on the ordering of the replies.

The Memory Unit takes memory instructions from the ROB with all operands resolved (the address and the value). To simplify the complexity of the Memory Unit, we require that the memory instructions must be sent in program order, and only after all previous branch instructions have been resolved. It is equally easy to express other more relaxed memory models in Bluespec. The Memory Unit makes any necessary memory accesses and returns the results to the ROB. Speculative stores must be kept until they are either invalidated or committed via two interfaces accessible by the ROB.

The ROB keeps track of the ordering of instructions it receives. It keeps track of which instructions are dependent on each other, and passes the values to instructions waiting for them. Whenever possible the ROB commits the oldest instructions which have been executed by writing the results back into the register file.

In this design, ROB unit also contains the Branch Unit. On branch misses it marks all the false path instructions as killed and increments the ROB's current epoch value. It also notifies the Fetch/Decode Logic of the correct program counter and the new epoch. Subsequent instructions which do not have the correct epoch will be thrown away when it is enqueued into the ROB.

We make the assumption that responses from functional units may not occur the same cycle as a request to the functional unit (i.e. there are no purely combinational functional units). There are no other timing requirements placed on the designing of the Fetch/Decode Logic.

3.2 Performance Goals

A reorder buffer should be able to simultaneously enqueue instructions, commit instructions, send instructions to the functional units, and receive responses from each functional unit. When designing in Verilog, this comes at the price of a horrific verification task. By using Bluespec, we can easily generate a correct circuit, but initially it may not perform all these tasks concurrently. In order to be acceptable our design must be able to achieve the same level of concurrency as is possible with handwritten RTL. We will examine if it is possible to transform the initial design such that the Bluespec compiler achieves the desired level of concurrency in the ROB module.

4 The Initial Design

This section details the work done to generate the initial design. The first implementation was the simplest natural way we could express the design in Bluespec.

Though the emphasis of this work is on cycle time performance, to be relevant the design must be realizable in hardware. As such, our design reflects appropriate high-level circuit considerations but ignores circuit-level opti-

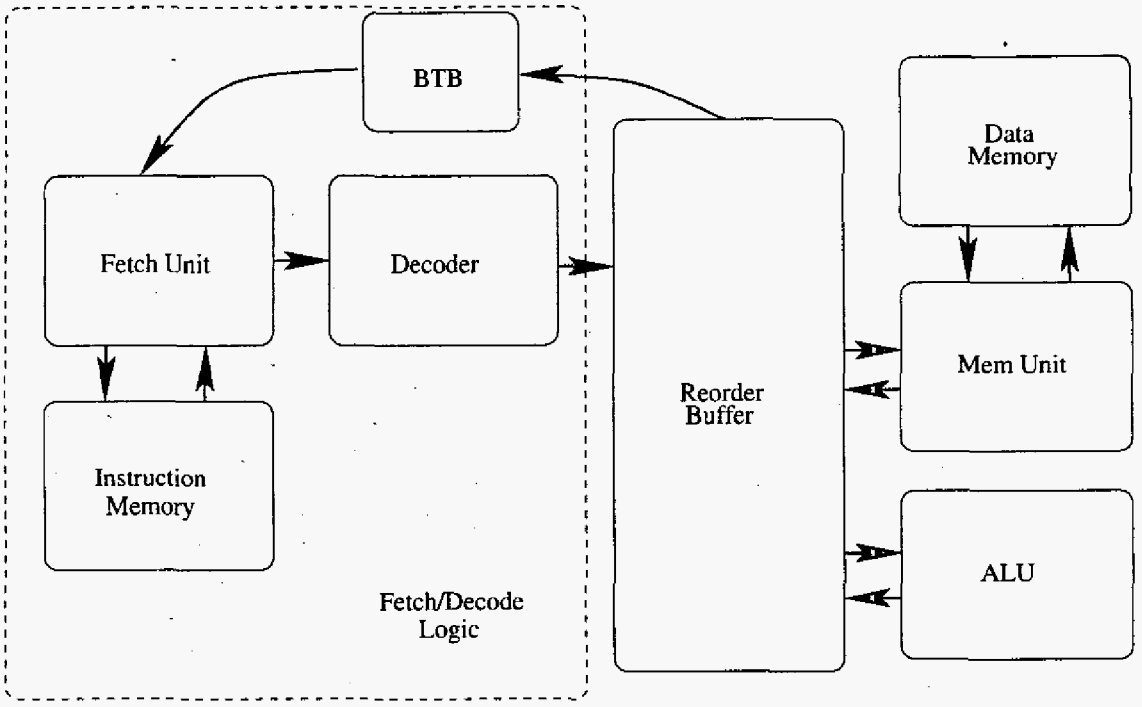

Figure 1. High Level Design of Processor

mizations, which can be performed after the RTL is generated.

4.1 Representation Considerations

First we considered how the ROB was going to interface with the rest of the processor. There are two general models for interaction. The first is a push model where we expect the module producing the data to pass it to the receiver. The second is a pull model, where the receiver grabs the result from the sender. The main difference between these functions is in which module the rule describing the data transfer will reside. Both methods will generate nearly identical hardware as the only change is in which module that particular action is placed.

To make the ROB module separately compilable, we adopted a combination of the two methods such that all interactions between the ROB and other modules could be performed by calling the methods of the ROB module. For example, the ALU gets the ready-to-execute ALU instructions from the ROB and process them. When it has completed an instruction it checks if the ROB can handle a response and if so, sends it to the ROB. Consider another interaction where the ROB must notify the Fetch/Decode Logic of a branch miss. To avoid the necessity of having the ROB call an interface method of the Fetch/Decode Logic, we added another interface method to the ROB which made the branch resolution information available. We also had to add a rule to the Fetch/Decode Logic to check for branch resolution updates. That is to say that on a branch miss we write to a register which can be accessed by the other modules through an interface. Thus, the Fetch/Decode Logic can look at this value, determine when the ROB had a branch miss and update the pc and epoch registers accordingly. This style handles all the interaction between the ROB and the other parts of the processor in a modular fashion.

We decided to keep track of the speculative state via a combinational lookup through the slots. This could also be done with an additional structure which kept the speculative value or tag reference of each register. The state would get copied during branch instructions and restored if the branch was mispredicted. Although this does offer a faster circuit length for insertions, we chose the combinational lookup, because the added circuitry would greatly increase the complexity of the design while only offering an improvement in the clock period which is not the main focus of this paper.

4.2 Storage

Instructions are kept in an ordered list of N slots. The slots contain the instruction and associated values required for execution, as well as the operand values, the result, and the slot's state. We use a headTag and a tailTag pointer to

represent respectively, the oldest slot used and the next slot in which an incoming instruction will be placed. To differentiate having the slot list full and empty we assert that one slot must remain empty.

```
struct Slot =
 tag :: ROBTag --the Slot's tag
 state :: Reg State
 ia :: Reg IA
 insType :: Reg InstrType
 opcode :: Reg (Bit osz)--opcode size
 tv1 :: Reg TagOrValue --operand 1
 tv2 :: Reg TagOrValue --operand 2
 imm :: Reg Imm -- immediate field
 dval :: Reg Value -- result
 destReg :: Reg RegOrHiLo
 predIa :: Reg PredIA --for branches
```

Each slot consists of a number of registers as shown below which represent an instruction template: the Instruction address (IA), the predicted instruction address (predIA), the slot's state, and two operand registers (tv1 & tv2) that store either the tag of the slot generating the value, or the actual value of the operand. We could have represented each slot as a single register, but by using a multiple register design, we help the compiler partition the data and generate better schedules.

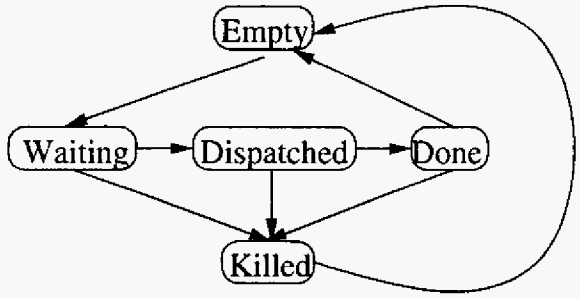

Figure 2. High Level Design of Processor

The state of a slot is either Empty, Waiting, Dispatched, Killed or Done. The state transition diagram is shown in Figure 2. Empty signifies that the slot has no instruction in it. Empty instructions only exist in the region that the headTag and tailTag denote as non-active. Upon having an instruction inserted into it, a slot enters the Waiting state where it will wait for its inputs to be resolved into actual values. After both inputs have been resolved the instruction in the slot can then be placed in the Dispatched state whenever the instruction is sent to the appropriate functional unit. When the result is sent back to the ROB and written into the slot, the slot enters the Done state where it can be committed and made Empty again. At any time, the branch resolution rule can set non-empty slot's state to Killed.

Instructions leave the ROB in the order they were inserted. To remove an instruction, one increments the headTag and writes the associated slot's state register as Empty. To insert an instruction, one increments the tailTag, places the instruction into the slot at which the tailTag pointed.

4.3 Design Complications

To match the MIPS I ISA we need to add a few additional complications to our design.

First, there is a branch delay slot. This means that when a branch instruction is killed we must keep the instruction directly after it. If we resolve the branch before this instruction has been inserted, the delay slot instruction will have the wrong epoch. To prevent this from happening we assert that branch instructions cannot be resolved until the next instruction has been inserted into the ROB.

Secondly, some instructions generate 64-bit results (i.e. multiply and divide instructions). To keep from having to double the size of the result in the slots, we place these instructions into consecutive slots with the high order bits in the first slot, and the low order bits in the second. The slots will then be treated as an atomic unit until the slots are committed.

4.4 The ROB Module

Below is a stylized description of our initial design. The sz value is a integer which the ROB is passed at instantiation. It represents the number of slots in the ROB. We can change this number to any value larger than 2 and maintain correctness.

```
mkROB :: Module (ROB sz)
mkROB sz = -- sz is # of slots
  module
   let
    minTag = 0
    maxTag = fromInteger sz
    --auxiliary functions
    --(e.g. mkSlot & incrTag)
    -- state elements
    rf :: RegFile <- mkRegFile
    curEpoch :: Reg Epoch <- mkReg 0
    headTag :: Reg ROBTag
                  <- mkReg minTag
    tailTag :: Reg ROBTag
                  <- mkReg minTag
    handlemissReg :: Reg (IA,PC,Epoch)
                        <-mkReg (0,0)
    slotList :: List Slot
     <- mapM (mkSlot) (upto minTag maxTag)
    rules
```

```
      <rules>
    interface
      enqueueInst inst = ...
      getALUInstr = ...
      getMEMInstr = ...
      updateALU tag result = ...
      updateMEM tag result = ...
      missvalues = ...
```

The enqueueInst interface does two combinational lookups to see if the two operands were generated by another instruction in the ROB and writes either the tag of the associated slot, or the value from the register file as appropriate into the operand registers and marks the slot as waiting to be dispatched (i.e. the state is Waiting).

```
enqueueInst inst =
   let
      --slot to write into
      slotJ = getSlot tailTag
      --structure with values to write
      slotVals = (getSlotValues inst)
   in
     action
        tailTag := incrTag tailTag
        writeSlot SlotJ slotVals
  when (not slotListFull)
```

There are two separate rules per slot which update the tagged values with the actual values. They look as follows:

```
"update TagOrValue i":
 when
     (T tag) <- slotJ.tv1
  ==> let
        slotTag = (getSlot tag)
      in
        action
          if (slotTag.state == Done) then
             slotJ.tv1 :=(V slotTag.dval)
           else
             noAction
```

This checks to see if the instruction in the slot (slotJ) associated with the given tag has been executed and if so, it writes the value into the operand register.

Additionally, for each slot there is a slot dispatch rule per functional unit type which takes ready waiting instructions and places them into the FIFOs which then dispatch to the appropriate functional units.

```
"Dispatch to ALU":
 when (slotJ.state == Waiting),
      (V v1) <- slotJ.tv1,
      (V v2) <- slotJ.tv2,
      (ALUTYPE == slotJ.instType)
  ==> let
        aluInst = (aluInstfromSlot slotJ)
      in
        action
          slotJ.state := Dispatched
          fifo2ALU.enq aluInst
```

As a side note, it may appear initially that generating these rules for each slot can be quite difficult and restrictive, but due to Bluespec's good static elaboration, the task is easily done. We do this by writing a function to generate rules for a single given slot. Then we can map this function over the slotList and concatenate the list of rules to our current list of rules for the ROB. This also gives us the additional benefit of not limiting the number of slots in the ROB.

```
let
  mkRules i = -- makes a slot's rules
    rules
      <rules>
 in
   mapM mkRules (upto minTag maxTag)
```

The interfaces to get the instruction from the ROB and hand it to the functional unit was a simple dequeue from the associated FIFO.

```
getALUInstr = do --actionValue
                 fifo2ALU.deq
                 return fifo2ALU.first
```

Branch instructions are executed by checking the result and killing all instructions after the branch and writing the register which is read by the Fetch Unit with the new pc and epoch value. These killed instructions are left in the list to be removed by the commit rule (i.e. the tailTag is not modified on a branch miss).

```
"Resolve Branch":
 when canFireBranch
  ==> let
        inst = fifo2branch.first
        correctIA = (calcNewIA inst)
        slotJ = (getSlot inst.tag)
      in
        fifo2branch.deq
        slotJ.state := Done
        if (correctIA /=
            inst.predIA) then
           action
             --send information on
             --branchmiss
             handlemissReg:=
```

```
            (correctIA,inst.IA,nextEpoch)
            curEpoch:=nextEpoch
        else
          noAction
```

The interface to get the new branch information just returns the value associated in the handlemissReg register.

```
missvalues = handlemissReg
```

Writebacks from the functional units write into the appropriate slot or slots.

```
updateMEM tag result =
   let
      slotJ = getSlot tag
    in
      action
         slotJ.state := Done
         slotJ.err   := result.err
         slotJ.dval  := result.value
```

Commits are done by removing the oldest instruction from the slot list and writing back any unkilled values to the register file.

```
"Commit":
 when headTag /= tailTag,
      slotJ <- getSlot headTag,
      slotJ.state == Done,
      not slotJ.err
  ==> action
         headTag := incrTag headTag
         slotJ.state := Empty
         (rf.write slotJ.destReg
          slotJ.dval)
```

5 Debugging the Design

After about a week of design and debugging we had the design completed. By this we mean that we were able to simulate the entire processor running MIPS I code using VCS.

After a 20 minute compile we found that the design had a CPI of 5 for a single dependency chain of ALU instructions. The reason for this is that it took one cycle for an instruction to be inserted into the ROB, one for an instruction to be committed, one to enter the instruction into the queue to the ALU, one to actually execute the instruction in the ALU, and one to propagate the executed value from one instruction to those requiring it.

This CPI is clearly unacceptable. These actions were designed to be completely disjoint (i.e. conflict free) from each other. As such, all the rules should be able to fire in parallel.

To determine why these rules conflict, we made use of the Bluespec compiler's rule conflict analysis function. First, we used the `-dschedule` flag to get a list of rule conflicts. This list shows which rules conflict with each other and which rule will be chosen to fire if both rules are enabled.

Once we have determined which rules conflict, we check to see whether the conflict was intentional. In the initial design resolving a branch miss conflicted with fetching a new instruction. This conflict is supposed to exist, as we do not want to fetch down a wrong path once we know we are on the wrong path. Other conflicts should not exist like the conflict between writing values back to the reorder buffer into two different slots from two different functional units. After identifying a pair of rules which we want to not conflict, we use the compiler's `-show-rule-rel` flag to get a list of state writes and reads each rule performs. From this list we can determine why the compiler believes the two rules to be conflicting. From there we can formulate the appropriate change to avoid the conflict.

6 Design Changes

¿From the initial design there were a number of problems which limited the performance of the design. These problems and their solutions are described in detail below.

6.1 Removing False Reads

The most common cause of false conflicts between rules was due to redundant clauses in the predicate. An example of this is the conflict between dispatches and the insert rule. All the dispatch rules had as part of their predicate a clause which checked to see if the slot was in the active area. This required that the tailTag needed to be read. This caused these rules to no longer be mutually exclusive with the insert rule which changes the tailTag value and limits sequential composability so that the insert rule would have to be simulated second. If the rules were only able to be sequentially composed in the opposite order naturally, it would cause false conflicts. The predicates were all rewritten to remove unnecessary references.

6.2 Improving Value Propagation Timing

In the original design, it took one cycle to propagate an executed value after execution from the generating slot to the slots waiting for the value. This was because the tags (tv1 & tv2) cannot be updated until the value has reached the slots they are referring to. For performance we must remove this 1 cycle delay.

We need to write into all the slots' operand registers when we are writing back the result. However, this causes a

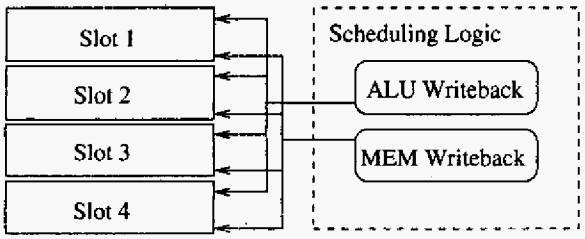

Figure 3. Conflict from Initial Design

huge number of possible writes which prevent two updates from being fired concurrently. This is also clearly not acceptable.

We would like to be able to split this rule into a rule per slot and thus avoid the unrealizable conflicts. However, because this is an interface method we cannot do so. For a Bluespec module to be able to be compiled modularly, the interface cannot be changed depending on its implementation.

To work around this restriction we need to use RWires. RWires are similar to Verilog wires, but with the added enabling bit for the signal exposed. That is we can view it as a register where writes are done before reads in a cycle with no ability to save values, which allows us to see when it has valid data. Using this allows us to know when values are being written and by doing so allows us to tailor what a rule does based on the other rules firing at the time. This must be used with caution, as it demands that you will now need to worry about some timing issues, but allows us to emulate some Verilog tricks which would otherwise be difficult.

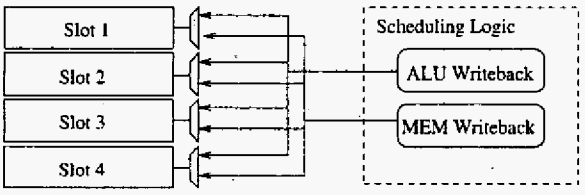

Figure 4. Solution with Mult. Update Ports

The initial solution was to make a special operand register module which would act as a register with multiple write ports (as in Figure 4). Writing into one of these ports would write into an RWire. Then every cycle, a rule in the module would fire and look at all the RWires in some fixed order, select the enabled value and write that into the register. Then each update unit could write into a different port and there would be no conflicts. We can guarantee that this is as correct as before, because there is only one value which an operand register will ever be written with, so it is not possible to miss any signals sent to the operand registers.

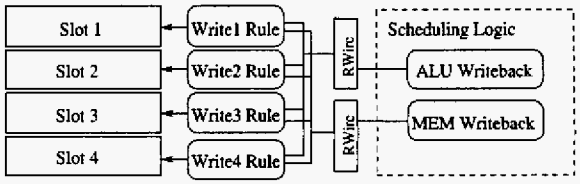

Figure 5. Solution Using RWires

Another solution, shown in Figure 5, was devised to improve the design's readability. Instead of having a separate RWire per slot per update rule, we only need one per update rule. Then the update rules can each read these values and do the updates to the register. This not only simplifies the description, but also allows us to make more complicated rule logics dependent on the state of the ROB more easily.

6.3 Removing State Register Conflicts

Another problem with the initial design was that a number of rules had false conflicts with each other. Analysis revealed that all the conflicts were caused by the rules writing to the slots' state registers. Automatic sequential composition by the compiler fails here, because the rules both needed to read and write the conflicting register, so composition is not possible. The only solution is to make the rules parallel in some way.

A natural thing to do is to add multiple prioritized write ports to each of these registers. However, we must be careful to give priority to the appropriate rules when two rules fire together.

The update, dispatch, insert, commit, branch rules all update the slots' state values. Upon initial inspection it is clear that the branch rule will be part of every rule conflict pair and that it must be prioritized higher than all other rules or possibly result in false path instructions not being killed. Thus we should only need two ports for the state registers.

Since during scheduling slot rules are treated as though they are using a method, if it is possible that they might use that method, we need to add more ports to handle the cases where the rules overlaps (e.g. a commit rule which takes the oldest N slots, and a insert rule which takes the next N slots, when there are less than N available slots, would have an overlap of at least one slot). We chose the order of highest precedence: the branch resolve rule, the commit rule, the update/writeback rules, the dispatch rules and lastly the insert rule. To make sure that any instructions inserted during a branch miss are not kept alive in the slot list we changed the resolve branch rule to also write the state of Empty into any slot which should not contain an active instruction to override the insert rule's values.

6.4 Reducing Dispatch Latency

In the initial design, an instruction had to first be put into a FIFO before it was dispatched into the associated functional unit. This introduces an extra cycle for each instruction to be able to pass its result along. We need this cycle to be removed.

To do this we must remove the intermediate FIFO. This means having some sort of combinational logic to get the slot which we want to take the instruction from so we can change its state. For each functional unit, we do a scan through the list of slots for valid instructions to send to that functional unit and pick the first one available. We do not attempt to search for the oldest executable instruction, because the hardware for this is very large. In Verilog designs, a similar simple static bias is used for instruction selection.

```
slotALU = findALUInst2Dispatch

getALUInstr =
   do
     -- write in 3rd port to avoid conflict
     slotALU.state._write3 Dispatched
     return (makeALUInstfromSlot slotALU)
   when validALUInst2Dispatch
```

This change introduces a conflict with the state register for each slot as now it is possibly writing into each state register. However, by adding another port to the state register as detailed in Section 6.3 we can avoid this conflict.

6.5 Reducing Branch Miss Penalty

When we added the extra interface to allow the ROB to be compiled separately from the rest of the design, we had to add a register on the path from the ROB to the Fetch/Decode Logic. This caused an increase of one cycle in the branch penalty. By replacing the register with an RWire we were able to remove this extra cycle.

7 Other Optimizations

In addition to the above improvements to the concurrency of the design, there are many simple improvements we can do which will reduce redundant hardware, and improve compilation. This section discusses the changes of this kind that we made to the design.

7.1 Reducing the Instruction Window Size

The first of such improvements we can make involves how we prefer which rules fire when given a choice between them. The best method is to favor completion of instructions over starting them. This will tend to reduce the amount of misspeculated instructions sent to the functional unit and reduce the number of cycles spent reclaiming killed slots.

Unfortunately, there is no way currently to explicitly encode a rule priority; the compiler selects what it believes to be a better preference weighting. However, in the case where the compiler has no preference it will tend to favor the rule first listed in the module description. By reorganizing the rules with this in mind we can achieve more appropriate prioritizations for the rules.

7.2 Reducing Conflicts

Some rules should never fire together, but they still are tested to see if they conflict. This can be quite expensive if the rules in question are both split into many pieces.

For instance, the rule to insert an instruction into a slot, and the rule which updates one of the tag values in that slot conflict as they both write to the same register (tv1 or tv2 depending on the particular rule). However, at a high level it is clear that the rules will never fire. By explicitly stating that the insert rule only operates on empty slots and the update on waiting slots, the compiler can very quickly determine the two rules are mutually exclusive. When this was done to the reorder buffer design, the compilation time was reduced by a factor of 20.

7.3 Improving Compilation

To remove some of the remaining issues, we need to split the interface rules (i.e. the insert rule). However, this cannot be done if we want a variable-length ROB. It would also expose the internals of the ROB modules, which we want to avoid. We hand-split the insert rule to operate on a per slot basis by having a rule for each slot which reads the input value from an RWire. The RWire is written to by the insert interface. We also needed to add a guard on interface by hand to make sure that it would only fire when there was a rule which would actually take the instructions put in the RWire. This may seem like a cycle sensitive change, but because we have already added multiple ports to the slot state registers, no additional conflicts are removed by this rule splitting. Instead it reduces the amount of checks the compiler has to do to determine that the rules do not conflict.

8 Findings

Within 100 man-hours we were able to generate our initial design, which was slow, but provably correct. After another 300 man-hours we were able to optimize our design so that it had a maximum theoretical CPI of 1 with a 2 wide insert/commit of the processor. This is 50% of

what we should be able to achieve. This inefficiency was due entirely to the one conflict we were unable to remove, which was between the commit rules and the insert instruction rules.

Our initial belief was that this was due to the problems with changing the headTag and the tailTag at once. However, the actual issue was with the register file and the headTag. The insert rule needs to read the headTag to verify if inserting is valid and the commit rule updates the headTag. Also, the commit rule writes back any unkilled rules it is committing and the insert rule reads the register file.

Thus each rule is writing something the other must read and so the rules conflict. Nevertheless, this is a false dependency. Firing the rules together always results in the same state as firing them on separate cycles, since any value being written into the register file is still contained in an active slot on that cycle. Thus the most current value would still be found by the insert rule if the rules were run concurrently.

In the course of our work, it was discovered that by limiting or splitting rules to act on as little state as possible and reducing the number of method conflicts between rules, we achieve better performance.

Unfortunately, this technique does not avoid this false read-write conflict for the current compiler. The Bluespec compiler is under revision with this problem in mind.

Bluespec has made much progress into making large scale designs. Many of the difficulties encountered in this work were due to needing to make changes in the design process to suit the language mindset, and understanding the conflict analysis of the compiler.

While our final CPI value was less than corresponding handwritten RTLs, steps to allow for directed relaxation of the conflict criteria will easily bridge this gap.

Work into the Bluespec compiler should now be directed towards automatically finding and safely integrating high-level knowledge into designs. Additionally, designing a concise way for the designer to describe how to handle two conflicting rules would be worthwhile task.

Acknowledgements

Funding for this research by the Defense Advanced Research Projects Agency under the IBM Contract NBCH3039004. Thanks is given to Professor Arvind and Daniel Rosenband for useful comments for both the design and paper; Mieszko Lis, Ravi Nanavati, Jacob Schwartz, and the rest of the Bluespec Inc. engineering team for their prompt help and compiler updates; and Karen Brennan for her editorial assistance.

References

[1] Arvind and X. Shen, *Using Term Rewriting Systems to Design and Verify Processors*, IEEE, Micro Special Issue on Modelling and Validation of Micro-processors Vol. 19(3): pp. 36-46, 1999.

[2] A. Benvensitee,P. Capsi,S. A. Edwards, N. Halbalbwachs, P. Le Guernic, and R. de Simone. *The Synchronous Languages 12 Years Later.* Proceedings of the IEEE, 91 (1). 64-93.

[3] D. L. Dill, *The Murphi Cerification System*, in Proceedings of the International Conference on Computer-Aided Design, Springer-Verlag, 1996.

[4] D. D. Gajski, *High Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic, Boston, 1992.

[5] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau, *SPARK "A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations"*, in International Conference of VLSI Design, (2003).

[6] G. Hajjiyiannis, S. Hanono, and S. Devadas *ISDL: An Instruction Set Description Language For Retargetability*, in Proceedings of the 34th Design Automation Conference (DAC), (1997), 299-302.

[7] J. C. Hoe, *Operation-Centric Hardware Description and Synthesis*, in Dept. of Electrical Engineering and Computer Science: Massachusetts Institute of Technology, 2000, p. 139.

[8] J. C. Hoe, and Arvind, *Synthesis of Operation-Centric Hardware Descriptions*, presented at IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2000.

[9] J. Plosila, and K. Sere, *Action Systems in Pipelined Processor Design*, in Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, (1997), 156-166.

[10] O. Schliebusch, A. Hoffman, A. Nohl, G. Braun, and H. Mayr, *Architecture Implementation Using the Machine Description Language LISA*, in Proceedings 7th Asia and South Pacific Design Automation Conference (ASP-DAC), (2002), 156-166.

[11] J. Straunstrup, and M. R. Greenstreet, *From High-Level Descriptions to VLSI Circuits*, BIT, 28 (3). 620-638.